Exploiting Opportunistic Physical Design in Large-scale Data Analytics

Jeff LeFevre[+*] Jagan Sankaranarayanan[*] Hakan Hacigumus[*]
Junichi Tatemura[*] Neoklis Polyzotis[+]
[*]NEC Labs America, Cupertino, CA [+]University of California Santa Cruz
{jlefevre,alkis}@cs.ucsc.edu, {jagan,hakan,tatemura}@nec-labs.com

ABSTRACT

Large-scale systems, such as MapReduce and Hadoop, perform aggressive materialization of intermediate job results in order to support fault tolerance. When jobs correspond to exploratory queries submitted by data analysts, these materializations yield a large set of materialized views that typically capture common computation among successive queries from the same analyst, or even across queries of different analysts who test similar hypotheses. We propose to treat these views as an opportunistic physical design and use them for the purpose of query optimization. We develop a novel query-rewrite algorithm that addresses the two main challenges in this context: how to search the large space of rewrites, and how to reason about views that contain UDFs (a common feature in large-scale data analytics). The algorithm, which provably finds the minimum-cost rewrite, is inspired by nearest-neighbor searches in non-metric spaces. We present an extensive experimental study on real-world datasets with a prototype data-analytics system based on Hive. The results demonstrate that our approach can result in dramatic performance improvements on complex data-analysis queries, reducing total execution time by an average of 61% and up to two orders of magnitude.

1. INTRODUCTION

Data analysts have the crucial task of analyzing the ever increasing volume of data that modern organizations collect and producing actionable insights. As expected, this type of analysis is highly exploratory in nature and involves an iterative process: the data analyst starts with an initial query over the data, examines the results, then reformulates the query and may even bring in additional data sources, and so on. Typically, these queries involve sophisticated, domain-specific operations that are linked to the type of data and the purpose of the analysis, e.g., performing sentiment analysis over tweets or computing the influence of each node within a large social network.

Not surprisingly, MapReduce (MR), be it the original framework, its open-source incarnation Hadoop or derivative systems such as Pig and Hive that offer a declarative query language, has become a de-facto tool for this type of analysis. Besides offering scalability to large datasets, MR facilitates incorporating new data sources, as there is no need to define a schema upfront and import the data, and provides extensibility through a mechanism of user-defined function (UDFs) that can be applied on the data. Since the computational scope of a single MR job is limited, scientists typically implement a query as an ensemble of MR jobs that feed data to each other. Quite often, such queries are written in a declarative query language, e.g., using HiveQL or PigLatin, and then automatically translated to a set of MR jobs.

Despite the popularity of MR systems, query performance remains a critical issue which in turn affects directly the “speed” at which data analysts can test a hypothesis and converge to a conclusion. Some gains can be achieved by reducing the overheads of MR, but the key impediment to performance is the inherent complexity of queries that ingest large datasets and span several MR jobs, a common class in practice. A-priori tuning, e.g., by reorganizing or preprocessing the data, is quite challenging due to the fluidity and uncertainty of exploratory analysis.

In this paper, we show that it is possible to dramatically improve query performance by leveraging the built-in fault-tolerance mechanism of MR as an *opportunistic physical design*. Specifically, we make the following observations:

- Each MR job involves the materialization of intermediate results (the output of mappers, the input of reducers and the output of reducers) for the purpose of failure recovery. More generally, a multi-stage job, such as one that is generated by Pig or Hive, will involve several such materializations. We refer to these materialized results as the artifacts of query execution and note that they are generated automatically as a by-product of query processing.

- Given the evolutionary nature of data exploration, it is likely that each query has similarities to previous queries by the same analyst, and even to queries of other analysts who examine the same data. For instance, several data analysts may perform sentiment analysis on a specific class of tweets (e.g., in a specific geographical area) but with a different hypothesis in mind. Hence, the computation performed by previous queries in the system, as captured in the generated artifacts, may be relevant for a new query.

Thus, we propose to treat artifacts as opportunistically-created materialized views and use them to rewrite a new query in the system. The opportunistic nature of our technique has several nice properties: the materialized views are generated as a by-product of query execution, i.e., without additional overhead; the set of views

is naturally tailored to the current workload; and, given that large-scale analysis systems typically execute a large number of queries, it follows that there will be an equally large number of materialized views and hence a good chance of finding a good rewrite for a new query. Our results with an implementation of this technique inside an industrial data-analytics system indicates that the savings in query execution time can be dramatic: a rewrite can reduce execution time by up to two orders of magnitude.

Rewriting a query using views is a well-studied problem in databases, yet its treatment in the context of MR involves a unique combination of technical challenges: there is a huge search space of rewrites due to the large number of materialized views in the opportunistic physical design; queries can be arbitrarily complex; and, views and queries almost certainly involve UDFs. Unfortunately, previous works do not address the problem in its full generality and essentially ignore one or more of the previous dimensions. Recent methods to reuse MR computations such as ReStore [3] and MR-Share [12] lack a semantic understanding of the artifacts produced during execution and can only reuse/share cached results when execution plans are identical. We strongly believe that any practical solution has to address the query rewrite problem in its full generality.

Contributions. In this paper we present a novel query-rewrite algorithm that targets the scenario of opportunistic materialized views within an MR system. The algorithm employs techniques inspired by spatial databases (specifically, nearest-neighbor searches in metric spaces [7]) in order to aggressively prune the huge space of candidate rewrites and generate the optimal rewrite in an efficient manner. Specifically, our contributions can be summarized as follows:

- A gray-box UDF model that is simple but expressive enough to capture many common types of UDFs. This affords us a limited understanding of UDFs to enable effective reuse of previous results. We provide the model and the types of UDFs it admits in Section 3.

- A rewriting algorithm that takes as input a query and a set of views and outputs the optimal rewrite. We show the algorithm is work-efficient in that it considers the minimal set of views necessary to find the optimal rewrite under certain assumptions. We describe this further in Section 5.

- Experimental results showing our methods provide execution time improvements up to two orders of magnitude using real-world data and realistic complex queries in Section 7. The savings from our method are due to moving much less data, avoiding the high expense of re-reading data from raw logs when possible, and reusing/repurposing results from long-running computations including UDFs.

2. PRELIMINARIES

Here we present the architecture of our system and briefly describe its individual components and how they interact, followed by our definitions.

2.1 System Architecture

Figure 1 provides a high level overview of the generic system framework we have developed. Our system integrates our query rewriting component with an existing query execution engine that is used by analysts.

We make the following assumptions about our target system and the nature of the queries. First, the optimizer takes a query written in some declarative language and translates it into an execution plan that consists of MR jobs. Second, queries are expressed against the base data, which are large logs and queries contain UDFs. Third, each MR job materializes its output to the stable storage (e.g., HDFS in Hadoop). Fourth, we assume that the optimizer can provide cost estimates for UDFs admitted to the system.

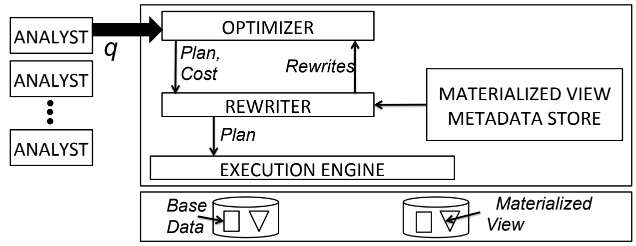


Figure 1: System diagram showing control flows

We extend the target execution engine by integrating our rewriter, presented in Section 5, as well as a materialized view metadata store. In order to let the rewriter communicate with the optimizer of the target engine, we extend the optimizer to generate a plan with two types of annotations on each plan node: (1) the logical expression of its computation and (2) the estimated execution cost.

The rewriter uses the logical expression in the annotation when searching for rewrites for the output of a node. The expression consists of relational operators or UDFs. For each rewrite found during the search, the rewriter utilizes the optimizer to obtain a plan and estimated cost.

During query execution, all by-products of query processing are retained as opportunistic materialized views, and stored in the system becoming part of its opportunistic physical design configuration. The materialized view metadata store contains information about the materialized views currently in the system such as the view definitions, and standard data statistics used in query optimization.

2.2 Notations

We use W to denote a plan generated by the query optimizer, represented as a DAG containing n nodes, ordered topologically. Each node represents an MR job. We denote the i^{th} node of W as NODE_i, $i \in [1, n]$. The plan has a single sink that computes the result of the query. Under the topological order assumption, this is NODE_n. W_i is a sub-graph of W containing NODE_i and all of its ancestor nodes. We refer to W_i as one of the rewritable *targets* of plan W. Following standard MR semantics, we assume that the output of each job is materialized to disk. Hence, a property of W_i is that it represents a materialization point in W. An outgoing edge from NODE_k to NODE_i represents data flow from k to i. V is the set of all existing materialized views (MVs) in the system.

The function $\text{COST}(W_i)$ takes as input W_i and returns its estimated cost provided by the query optimizer. $\text{COST}(\text{NODE}_i)$ is the cost of the MR job at NODE_i, given its input data. $\text{COST}(W_i)$ $= \sum_{\forall \text{NODE}_k \in W_i} \text{COST}(\text{NODE}_k)$.

We use r_i to denote an equivalent rewrite of target W_i using only views in V. r_i is considered an equivalent rewrite of W_i *iff* it produces an identical output to W_i given the same database instance D. A rewrite r^* represents the minimum cost rewrite of W (i.e., W_n).

L_W represents the language of the queries, which is the declarative query language used by an analyst when submitting a query to our system. This is also the language of the views. L_W includes all the relational operators in this query language, as well as all the UDFs admitted to the system by our UDF model (presented in

Section 3).

L_R is a subset of the declarative query language used by the analyst (i.e., $L_R \subseteq L_W$), and it represents the language used for rewriting queries. L_R includes select, project, join, group-by, and aggregation (SPJGA). Our system provides an interface to easily add UDFs to L_W, but every UDF admitted to the system should not be added to L_R. Although our techniques need not be changed, adding UDFs to L_R significantly increases the complexity of the search for rewrites. The implication of $L_R \subseteq L_W$ is that it may not be possible to find an equivalent rewrite of a query in L_W using L_R and the available views.

2.3 Problem Definition

Given these basic definitions, we introduce the problem we solve in this paper.

Problem Statement. *Given a plan W for an input query q, and a set of materialized views V, find the minimum cost rewrite r^* of W.*

Our rewrite algorithm considers views in V and the rewrite language L_R to search for r^*. Since some views may contain UDFs, we require an understanding of UDFs in order for the rewriter to utilize those views during its search. Next we will describe our UDF model and then present our rewrite algorithm that solves this problem.

3. UDF MODEL

Queries in our scenario are likely to include complex analytical operations expressed as UDFs. In order to reuse previous computation in our system effectively, we require a way to model UDFs semantically.

Possibilities for modeling UDFs may include white, gray, or black-box approaches with varying levels of overhead and complexity to the system. A white box approach requires a complete description of the UDF such that the system understands *how* the UDF transforms the inputs. This approach has high overhead for an analyst when adding a new UDF to the system. A black box approach has very low overhead for an analyst but produces an output that is completely opaque to the system thus may not be suitable for our goal of reusing results. Since UDFs can involve fairly complex operations on the data, our system adopts a gray-box approach that only captures the *end-to-end transformations* performed by a UDF. By *end-to-end transformations*, we imply that our model can capture fine-grain dependencies between the input and output tuples, although our UDF model does not know anything more about the computation. This requires additional work to provide the gray-box model when adding a new UDF, yet allows the system to understand the UDF's transformations in a useful way. A black-box model, on the other hand, can only track coarse-grain dependency between input and output as a whole.

A UDF in our gray-box model is written as a *composition* of *local functions*. A local function refers to a function that operates on a single tuple or a single group of tuples. We restrict a local function to perform the following operations.

1. Discard or add attributes
2. Discard tuples by applying filters
3. Perform grouping of tuples

The gray-box model does not understand the nature of the transformation performed by the local functions however it understands the transformation afforded by each local function. The end-to-end transformation of the UDF can be obtained by composing the operations performed by each local function in the UDF

Following our gray-box model, the input and output of every node in the plan is captured by three properties: attributes A, filters F, and group-by G. F is the conjunction of all filters applied to the input data, G is the current grouping applied, and A captures the schema. The end-to-end transformation of a UDF can be expressed as the transformation of the input to the output using the composition of the local functions. Note that the composition captures the *semantics* of the end-to-end transformation using the three operations, but not the actual computation. and not to describe the internal procedure. By combining these with grouping, our model can express rich UDFs, as well as relational operators such as select, project, join, group-by and aggregation. Joins are modelled in the standard way in MapReduce which is to group multiple relations on a common key (as in Pig [15]).

3.1 UDF Example

The following example illustrates how the model captures one possible implementation of a UDF called `FOOD-SENTIMENT` that uses a classifier to compute a sentiment score on users who talk about food items.

EXAMPLE 1. *`FOOD-SENTIMENT` UDF is expressed as a composition of the following local functions: For each user, gather all of their tweets from the Twitter log; Apply a classifier that determines if a tweet refers to food or a closely-related concept, and then computes a food sentiment score for each user; Output only those users with score above a given threshold,* 0.5. *The representation of this UDF in the gray-box model is as follows. Suppose the inputs are defined as attributes A={`userid`, `tweet`}, filters F=∅, group-by G=∅, and the UDF name is `FOOD-SENTIMENT`. Then the output includes the new attribute `score` and an additional filter, given by A'={`userid`, `score`}, F'={`score`>0.5}, and G' ={`userid`}.*

Note from the three operations described, there is no facility to update attributes in place, which is reasonable given that our primary data source is logs. Any modification to an input attribute must be reflected as a new attribute. When a new attribute is produced by a UDF, its dependencies on the subset of the input are recorded as well as the UDF name. For instance, the meta-data describing the new attribute `score` in Example 1 is: ⟨A={`userid`, `tweet`}, F=∅, G=∅, name=`FOOD-SENTIMENT`⟩.

Any UDF with end-to-end transformations correctly captured by the model can be admitted to the system. We use this model to capture UDFs such as classifiers, lat/lon extractor, sentiment analysis, log extractor and others as we show later in our experimental evaluation. Note the model only needs to be provided the first time the UDF is added. This gray-box model has certain limitations. For instance, given two UDFs with different names but identical inputs and functionality, the new attributes they produce will have different meta-data. This deficiency can only be overcome by resorting to a white-box approach.

3.2 Cost of UDF

We now briefly describe how the optimizer costs a UDF given its gray-box description and its composition in terms of local functions. We assume that a mapping between each local function to the number of MR jobs, map and reduces phases are known to the optimizer. The optimizer costs a UDF by summing up the cost of performing the local functions to reasonably estimate cost.

Suppose that a local function performs a set S of operations given by the three UDF model. Without a mapping of the operations to the number of jobs, maps and reduce, we use the following weak property of a cost optimizer.

DEFINITION 1. *Non-subsumable cost property: Let S be any set of operations represented by the UDF model. Let $\text{Cost}(S, D)$ be defined as the total cost of performing all operations in S on a database instance D. Let $\text{Cost}(x, D) = \infty$ for those operations $x \in S$ that cannot be applied on D. Then the following property states the cost of performing S on a database instance D is at least as much as performing the cheapest operation in S on D.*

$$\text{Cost}(S, D) \geq \min(\text{Cost}(x, D), \forall x \in S)$$

Because the lower bound is derived from the minimum operation cost, it provides a weak lower bound. A tighter bound might be provided if we could use the cost of the most expensive operation in S, i.e., $\max(\text{Cost}(x, D), \forall x \in S)$. The reason we must rely on the weak min property is because a stronger max property requires $\text{Cost}(S', D) \leq \text{Cost}(S, D)$, where $S' \subseteq S$. This requirement is difficult to meet in practice. As a simple counterexample, suppose S contains a filter with high selectivity and a group-by operation with higher cost than applying the filter, when considering the operations independently on database D. Let S' contain only group-by. Suppose that applying the filter before group-by results in few or no tuples streamed to group-by. Then applying group-by can have nearly zero cost and it is plausible that $\text{Cost}(S', D) > \text{Cost}(S, D)$.

4. PROBLEM OVERVIEW

Two factors make our problem challenging: (1) the complexity of answering queries using views, and (2) the property that $L_R \subseteq L_W$.

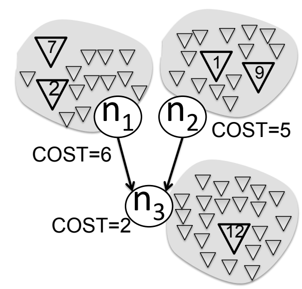
First, answering queries using views is a long-studied problem [1, 6, 11] and known to be hard. When both queries and views are expressed in a language that only includes conjunctive queries, it is known to be NP-complete. A *complete* rewriting of a query using views is one that only uses views and does not use any of the base relations. Determining whether there is a complete rewriting of a query using a set of views is also NP-complete [11] for conjunctive queries. In this work we consider only complete rewrites.

When searching for a rewrite of a target, existing views in V may only represent *partial* solutions [6] with respect to the target. Partial solutions can be repeatedly merged to produce *complete* solutions, which explodes the space further [2, 6]. *Candidate views* are those considered by the rewriter, which includes all views in V and all hypothetical views obtained by merging views in V during the process of creating complete solutions. Finding the minimum cost equivalent rewrite of W requires considering merging views in V and operations in L_R, making the solution search space exponential both in 1) the number of views in V and 2) the size of L_R. Thus finding a rewrite for a single target in W is computationally expensive.

Second, as noted in Section 2.2, $L_R \subseteq L_W$ has the following implication that make a rewrite algorithm more challenging. Searching for an optimal rewrite of only W_n does not suffice. This is because the inability to find a rewrite for W_n does not mean that one cannot find a rewrite at a different target in W. If a rewrite r_i is found for W_i, r_n can be expressed as a rewrite for W_n by combining r_i with the remaining nodes in W indicated by $\text{NODE}_{i+1} \cdots \text{NODE}_n$. Thus the search process must happen at all targets in W.

A straightforward solution is to search for rewrites at all targets of W. It obtains the best rewrite for each target, if one exists, and chooses a subset of these to obtain r^* using a dynamic programming approach. One drawback of this approach is that there is no easy way of early terminating the search at a single target. Another drawback is that even with an early termination property, the algorithm may search for a long time at a target (e.g., W_i) only to find an expensive rewrite, when it could have found a better (lower-cost) rewrite at an upstream target (e.g., W_{i-1}) more quickly. This is illustrated in Example 2.

EXAMPLE 2. *W contains 3 nodes, n_1, n_2, n_3, each with their individual costs as indicated, where cost of $W = 13$. Alongside each node is the space of views to consider for rewriting. Candidate views that result in a rewrite are indicated by triangles with the rewrite cost and those that fail to yield a rewrite are indicated by the empty triangles. An algorithm could have examined all the views at W_3 finally identifying the rewrite with a cost of 12. However, as noted, this algorithm cannot stop. Notice that by searching for rewrites of W_1 and W_2, the algorithm would find a rewrite of cost 2 for W_1, and of cost 1 for W_2. It could then combine these with the cost of 2 for the node n_3, resulting in a rewrite of W_3 with a total cost of 5. This is much less than the rewrite found for W_3 with a cost of 12. Had the algorithm known about the low-cost rewrites at n_1 and n_2, it could have searched less of the space at n_3.*



Now assume that, during the search for rewrites of a given target, we can somehow obtain the lower bound of the cost of possible rewrites. Such information would be useful to stop the search earlier in two ways. First, if we know the lowest cost of rewrites in the unexplored space of a single target is higher than the cost of a rewrite already found, there is no use to continue searching the remaining space. Second, the failures and successes at finding a rewrite at one target can be used to inform the search at a different target. For example, if we already found the best rewrites for n_1 and n_2, we can stop search at n_3 when all remaining rewrites for n_3 have a cost greater than 5. Assuming that lower bounds are available, a challenging question then is how to best utilize such information to search this complex space of query rewriting.

4.1 Outline of our Approach

We propose a work efficient query rewriting algorithm that searches the space at each target ordered by a lower bound on the cost of a rewrite using a view. The lower bound should not require finding a valid rewrite, as this is computationally expensive. Thus we wish to utilize an alternative cost function that is easy to compute and has some desirable property with respect to the actual cost of a rewrite.

We define an optimistic cost function $\text{OptCost}(W_i, v)$ that takes a candidate view v and target W_i as inputs and provides a lower-bound on a rewrite r_i of W_i using v. r_i is a rewrite of W_i that uses the candidate view v. The property of the lower bound is

$$\text{OptCost}(W_i, v) \leq \text{Cost}(r_i)$$

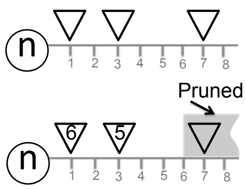
which we will describe further in Section 6.2. For the purposes of discussion here we assume the existence of this function and it can be employed as a black-box. The use of a lower-bound cost is inspired by nearest neighbor finding problems in metric spaces where computing distances between objects can be computationally expensive, thus preferring an alternate distance function that is easy to compute with the desirable property that it is always less than or equal to the actual distance.

Given OPTCOST function, our rewrite algorithm finds the optimal rewrite r^* of W by breaking the problem into two components:

1. BFREWRITE (Section 5) performs an efficient search of rewrites for all targets in W and outputs a globally optimal rewrite for W_n.

2. VIEWFINDER (Section 6) enumerates candidate views for a single target based on their potential to produce a low-cost rewrite of the target, and is utilized by BFREWRITE.

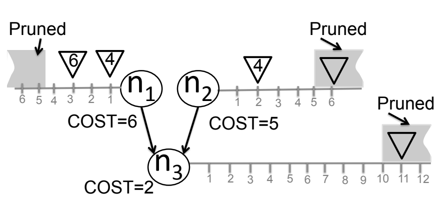
The VIEWFINDER orders candidate views at a target as illustrated in the example alongside. These graphics show candidate views as triangles arranged by their OPTCOST to a target as indicated by the scaled line below the views. The OPTCOST of all views can be obtained quickly, which determines their placement along the line. The top graphic shows the initial state of VIEWFINDER before searching for rewrites. The bottom graphic shows the state of VIEWFINDER after examining two candidate views for rewrite. The cost of the rewrite for each candidate view is indicated inside the triangle. The VIEWFINDER began by examining the view at OPTCOST of 1 as this view had the potential to produce the lowest-cost rewrite of the target. It obtains the actual cost of a rewrite using this view, found to be 6 as marked. The next candidate view was at OPTCOST of 3, which is examined and its rewrite cost was found to be 5. So, the best rewrite found for the target is now 5. The utility of the lower bound cost can be immediately seen now as there is no reason to examine the next candidate view which has an OPTCOST of 7. Thus any remaining views can be pruned from consideration, and the VIEWFINDER at this target can terminate early.



The BFREWRITE spawns one instance of VIEWFINDER per target in W and uses the information from the search at a target to inform and control the search at other targets. By exploiting OPTCOST our algorithm performs an incremental search with a VIEWFINDER at each target. It starts with the candidate views at different targets that have the potential to form the lowest cost rewrite of W and continues its search in an incremental fashion until termination. In this way our approach obviates the need to look in sub-spaces where it is not possible to find a cheaper rewrite for W. This forms the *best-first* nature of our solution. Note that due to the known hardness of the problem, in the worst case it must examine all candidate views at each target.

To illustrate BFREWRITE process, the example shown alongside again shows W containing 3 targets n_1, n_2 and n_3 with cost 6, 5, and 2, respectively. The candidate views at each target are ordered by their OPTCOST value as described previously.

Our best-first algorithm would first examine a rewrite composed of the next candidate view at n_1 (OPTCOST of 1), the next candidate view at n_2 (OPTCOST of 2) and node n_3 (COST of 2). This combination has the *potential* to produce a rewrite of W_3 with an OPTCOST of 1+2+2=5. However, upon obtaining the actual rewrite of the candidate views at n_1 and n_2, we can see that the actual cost of this rewrite is 10. The algorithm continues by examining the next candidate view at n_1 (OPTCOST of 3). This can potentially produce a rewrite of W_3 by composing: OPTCOST of 3 at n_1, plus rewrite with COST 4 at n_2, plus n_3 with COST 2 (3+4+2=9). However, this candidate view at n_1 was found to have an actual cost of 6. Hence, this rewrite of W has a COST of 12 (6+4+2) and can be discarded since BFREWRITE already has found a rewrite of W with COST 10. The algorithm need not examine any remaining candidate views in n_1 and n_3 as they do not have the potential to produce a cheaper rewrite.



5. BEST-FIRST REWRITE

The BFREWRITE algorithm produces a rewrite r of W that can be composed of rewrites found at multiple targets in W. The computed rewrite r^* has provably the minimum cost among all possible rewrites in the same class. Moreover, the algorithm is work-efficient: even though COST(r^*) is not known a-priori, it will never examine any candidate view with OPTCOST higher than the optimal cost COST(r^*). Intuitively, the algorithm explores only the part of the search space that is needed to provably find the optimal rewrite.

The algorithm begins with W itself being the best rewrite for the plan. It then spawns n concurrent search problems at each of the targets in W and works in iterations to find a better rewrite. In each iteration, the algorithm chooses one target W_i and examines a candidate view at W_i. The algorithm makes use of the result of this step to aid in pruning the search space of other targets in W. To be work efficient, the algorithm must choose correctly the next candidate view to examine. As we will show below, the OPTCOST functionality plays an essential role in choosing the next target to refine.

The BFREWRITE uses an instance of the VIEWFINDER to search the space of rewrites at each target. We will describe the details of VIEWFINDER in Section 6. In this section, VIEWFINDER is a black box that provides the following functions: (1) INIT creates the search space of candidate views ordered by their OPTCOST, (2) PEEK provides the OPTCOST of the *next* candidate view, and (3) REFINE tries to find a rewrite of the target using the next candidate view. One important property of REFINE is the following: there are no remaining rewrites to be found for the corresponding target that have a cost less than the value of PEEK.

5.1 Algorithm

In this section, we present the details of our BFREWRITE algorithm and in the following section we provide an example that illustrates how the algorithm works on a small instance of the problem.

Algorithm 1 Optimal rewrite of W using VIEWFINDER

function BFREWRITE(W, V)
    for each $W_i \in W$ **do** ▷ Init Step
        VIEWFINDER.INIT(W_i, V)
        BSTPLN$_i$ $\leftarrow W_i$ ▷ original plan to produce W_i
        BSTPLNCST$_i$ $\leftarrow$COST(W_i) ▷ plan cost
    end for
    repeat
        $(W_i, d) \leftarrow$ FINDNEXTMINTARGET(W_n)
        REFINETARGET(W_i) **if** $W_i \neq$ NULL
    until $W_i =$ NULL
    Rewrite W using BSTPLN$_n$
end function

Algorithm 1 presents the main BFREWRITE function, which primarily repeats the following procedure: Choose the best target W_i

(by FINDNEXTMINTARGET) and refine it by asking VIEWFINDER to examine the next candidate view (using REFINETARGET).

For each target $W_i \in W$, the algorithm maintains the best rewrite BSTPLN$_i$ of W_i found so far and its cost BSTPLNCST$_i$ (which are initialized in lines 2–6). The main loop iterates the search procedure until there is no target that can possibly improve BSTPLN$_n$, at which point r^* has been identified.

In line 8, the return value W_i of FINDNEXTMINTARGET corresponds to the next target to continue searching, while d represents the minimum OPTCOST of a rewrite for W_n involving a candidate view at W_i that has not been examined so far. As we will see shortly, FINDNEXTMINTARGET() examines views in increasing OPTCOST order at each target and so can guarantee that the return value d can never decrease. This property ensures that BFREWRITE has examined all possible rewrites for W with actual cost less than d, which in turn allows for early termination when the cost of the currently best rewrite is not greater than d.

Algorithm 2 Find next min target to refine

1: **function** FINDNEXTMINTARGET(W_i)
2: $d' \leftarrow 0$; $W_{MIN} \leftarrow$ NULL; $d_{MIN} \leftarrow \infty$
3: **for each** incoming vertex NODE$_j$ of NODE$_i$ **do**
4: $(W_k, d) \leftarrow$FINDNEXTMINTARGET(W_j)
5: $d' \leftarrow d' + d$
6: **if** $d_{MIN} > d$ and $W_k \neq$ NULL **then**
7: $W_{MIN} \leftarrow W_k$
8: $d_{MIN} \leftarrow d$
9: **end if**
10: **end for**
11: $d' \leftarrow d' +$ COST(NODE$_i$)
12: $d_i \leftarrow$ VIEWFINDER.PEEK()
13: **if** $\min(d', d_i) \geq$ BSTPLNCST$_i$ **then**
14: **return** (NULL, BSTPLNCST$_i$)
15: **else if** $d' < d_i$ **then**
16: **return** (W_{MIN}, d')
17: **else**
18: **return** (W_i, d_i)
19: **end if**
20: **end function**

FINDNEXTMINTARGET, given in Algorithm 2, identifies the next target to be refined in W, as well as the minimum cost (OPTCOST) of a possible rewrite for W_i. There can be three outcome of a search at a target W_i. Case 1: W_i and all its ancestors cannot provide a better rewrite. Case 2: An ancestor target of W_i can provide a better rewrite. Case 3: W_i can provide a better rewrite. By recursively making the above determination at each target in W, the algorithm identifies the target to refine next.

The algorithm finds the best rewrite obtained by combining rewrites found at the ancestors of W_i. This rewrite has an OPTCOST of d', which is acquired by summing the VIEWFINDER.PEEK values at the ancestors of W_i using the recursive procedure and the cost of NODE$_i$ (lines 3–11). Note that we also record the target W_{MIN} representing the ancestor target with the minimum OPTCOST candidate view (lines 6–9). Next, we assign d_i to the next candidate view at W_i using VIEWFINDER.PEEK (line 12).

Now the algorithm deals with the three cases outlined above. If both d' and d_i are greater than or equal to BSTPLNCST$_i$ (case 1), there is no need to search any further at W_i (line 13). If d' is less than d_i (line 15), then W_{MIN} is the next target to refine (case 2). Else (line 18), W_i is the next target to refine (case 3).

Algorithm 3 describes the process of refining a target W_i. Refinement is a two-step process. First it obtains a rewrite r_i of W_i from VIEWFINDER if one exists (line 2). The cost of the rewrite r_i obtained by REFINETARGET is compared against the best rewrite

Algorithm 3 Queries VIEWFINDER in best-first manner

1: **function** REFINETARGET(W_i)
2: $r_i \leftarrow$VIEWFINDER.REFINE(W_i)
3: **if** $r_i \neq$ NULL and COST$(r_i) <$ BSTPLNCST$_i$ **then**
4: BSTPLN$_i \leftarrow r_i$
5: BSTPLNCST$_i \leftarrow$COST(r_i)
6: **for each** edge (NODE$_i$, NODE$_k$) **do**
7: PROPBSTREWRITE(NODE$_k$)
8: **end for**
9: **end if**
10: **end function**

1: **function** PROPBSTREWRITE(NODE$_i$)
2: $r_i \leftarrow$plan initialized to NODE$_i$
3: **for each** edge (NODE$_j$, NODE$_i$) **do**
4: Add BSTPLN$_j$ to r_i
5: **end for**
6: **if** COST$(r_i) <$ BSTPLNCST$_i$ **then**
7: BSTPLNCST$_i \leftarrow$COST(r_i)
8: BSTPLN$_i \leftarrow r_i$
9: **for each** edge (NODE$_i$, NODE$_k$) **do**
10: PROPBSTREWRITE(NODE$_k$)
11: **end for**
12: **end if**
13: **end function**

found so far at W_i. If r_i is found to be cheaper, the algorithm suitably updates BSTPLN$_i$ and BSTPLNCST$_i$ (lines 3–9). In the second step (line 7), it tries to use r_i as part of a rewrite of W using the recursive function given by PROPBSTREWRITE in Algorithm 3. After this two-step refinement process, BSTPLN$_n$ contains the best rewrite of W found so far.

The recursion procedure given by PROPBSTREWRITE pushes downward the new BSTPLN$_i$ along the outgoing nodes and towards NODE$_n$. At each step it composes a rewrite r_i using the immediate ancestor nodes of NODE$_i$ (lines 2–5). It compares r_i with BSTPLN$_i$ and updates BSTPLN$_i$ if r_i is found to be cheaper (lines 6–12).

5.2 Algorithm Example

Example 3 shows how BFREWRITE attempts to rewrite a sample plan.

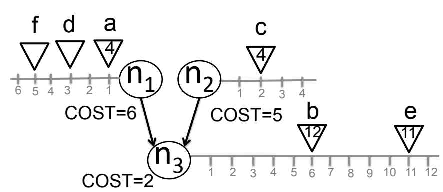


Figure 2: Three targets n_1, n_2 and n_3 with candidate views ordered by their OPTCOST a–f.

EXAMPLE 3. *Figure 2 shows a plan containing three nodes n_1, n_2 and n_3, where n_3 represents W_n. The five candidate views a–f are arranged by their* OPTCOST *from the target nodes. For example, b is placed at* OPTCOST *of 6 for node n_3. The cost of nodes n_1, n_2, n_3 is 6, 5, 2 respectively, hence* BSTPLN$_3$ *(i.e.,*BSTPLN$_n$*) begins with* COST*=13.*

In the first step of the algorithm, the cheapest potential rewrite for n_3 is determined to be $(a = 1)$+$(c = 2)$+$(n_3 = 2)$ having a total OPTCOST *of 5, whereas the next cheapest possible rewrite of n_3 using b has an* OPTCOST *of 6.* FINDNEXTMINTARGET *chooses a as the first candidate view to* REFINE *within $a + c + n_3$ because the* OPTCOST *of a (=1) is less than c (=2). After refining a, the actual* COST *of a is found to be 4, which is shown by the dashed line from a in Figure 2. Therefore* BSTPLNCST$_1$ *is set to*

4. Now since the best known rewrite for n_3 is $(a = 4)$+$(n_2 = 5)$+$(n_3 = 2)$ is 11, the value BSTPLNCST$_3$ *is updated from 13 to 11 by* PROPBSTREWRITE.

Next it attempts the rewrite for n_3 using b with an OPTCOST *of 6, which is less than the next best choice of $(d = 3)$+$(c = 2)$+$(n_3 = 2)$ with a total* OPTCOST *of 7. The* COST *of the rewrite of n_3 using b was found to be 12 as indicated in Figure 2. Since* BSTPLNCST$_3$ *is already 11, it is not updated.*

The next best choice for n_3 is the rewrite with the OPTCOST *of 7. Within that rewrite,* FINDNEXTMINTARGET *chooses to refine c, yielding a rewrite for n_2 whose actual cost is 4, so* BSTPLNCST$_2$ *is set to 4. Then* BSTPLN$_3$ *and* BSTPLNCST$_3$ *are set to be $(a = 4)$+$(c = 4)$+$(n_3 = 2)$ with a total cost of 10. Figure 2 captures this present state of the algorithm after the first three rewrite attempts.*

The algorithm would proceed to the next best possible rewrite of $(d = 3)$+$(c = 4)$+$(n_3 = 2)$ with OPTCOST *of 9 which is still better than the best known rewrite of n_3 with cost of 10. The algorithm terminates when there are no possible rewrites remaining for W_n with a* OPTCOST *less than* BSTPLNCST$_n$. *Any view with an* OPTCOST *greater than their target node's* BSTPLNCST *can be pruned away, e.g., e at n_3 (since $11 > 10$) and f at n_1 (since $5 > 4$).*

It is noteworthy in Example 3 that had the algorithm started at n_3 first, it would have examined all the candidate views of n_3, resulting in a larger search space than necessary.

5.3 Proof of Correctness

The following theorem provides the proof of correctness and the work efficiency property of our BFREWRITE algorithm.

THEOREM 1. *The* BFREWRITE *finds the optimal rewrite r^* of W and is work efficient.*

PROOF. Finding the optimal rewrite requires that the algorithm must not terminate before finding r^*. Ensuring work efficiency requires that the algorithm should not examine any candidate views that cannot be possibly included in r^*. First we show these properties for a W containing a single target, then provide an outline of how to extend these results to an arbitrary plan containing n targets. For the single target case, the algorithm must examine every candidate view with OPTCOST less than or equal to COST(r^*), but must not examine any candidate view with OPTCOST greater than COST(r^*).

Proof by contradiction is as follows. Suppose that the algorithm found a candidate view v, which resulted in a rewrite r. Suppose that the candidate view v^* which produces the optimal rewrite r^* is not considered by the algorithm before terminating thus incorrectly reporting r as the optimal rewrite even though COST(r^*) < COST(r). As the candidate view v was examined before v^* from the PQ, it must have been the case that OPTCOST(v) < OPTCOST(v^*). Once v was found BSTPLNCST$_n$ is set to COST(r), which means the algorithm will not terminate until all candidate views whose OPTCOST is less than or equal to BSTPLNCST$_n$ (i.e., COST(r)) have been found. Combining the above inequalities with the lower bound property of OPTCOST with respect to COST, we have:

OPTCOST(v) < OPTCOST(v^*) ≤ COST(r^*) < COST(r) = BSTPLNCST$_n$.

This means that the algorithm must have examined v^* (and consequently r^*) before terminating. Hence the contradiction that r was reported as the best rewrite.

Consider the general case of W containing n targets. It suffices to show that the algorithm, using FINDNEXTMINTARGET, reduces the n priority queues to an equivalent problem of a single global priority queue. The queue contains all possible rewrites that include all candidate views from every target, and each rewrite is ordered by its OPTCOST. This reduction is straightforward and omitted due to lack of space. □

6. VIEWFINDER

The key feature of VIEWFINDER is its OPTCOST functionality which is used by the BFREWRITE to explore the space in an incremental manner and prune unnecessary sub-spaces as shown in Section 4.1. As noted earlier, rewriting queries using views is known to be a hard problem. Traditionally, methods for rewriting queries using views for the class of SPJG queries use a two stage approach [2, 6]. The prune stage determines which views are *relevant* to the query, and among the relevant views those that contain all the required join predicates are termed as *complete*, otherwise they are called *partial* solutions. This is typically followed by a merge stage that joins the partial solutions using all possible equijoin methods on all join orders to form additional relevant views. The algorithm repeats until only those views that are useful for answering the query remain.

We take a similar approach in that we identify partial and complete solutions, then follow with a merge phase. The VIEWFINDER considers candidate views C when searching for rewrite of a target. C includes views in V as well as views formed by "merging" views in V using a MERGE function, which is an implementation of a standard view-merging procedure (e.g., [2, 6]). Traditional approaches merge all partial solutions to create complete solutions, and continues until no partial solutions remain. This "explodes" the space of candidate views exponentially. Our approach allows for a gradual explosion of the space as needed, which results in far fewer candidates views from being considered.

With no early termination condition existing approaches would have explore the space exhaustively at all targets. Thus we desire a rewriting algorithm that can enumerate the space and incrementally explore only as much as required, frequently stopping and resuming the search as requested by BFREWRITE. We note that while an equivalent rewrite for a target may exist, the VIEWFINDER may never be asked to find it, as illustrated by Example 2 for the case of the candidate view with cost 12 at n_3.

6.1 Algorithm

The VIEWFINDER is presented in Algorithm 4. At a high level, the VIEWFINDER is *stateful* which enables the BFREWRITE to start, stop and resume the incremental searches at each target. The VIEWFINDER maintains state using a priority queue of candidate views. The VIEWFINDER implements three functions INIT, PEEK and REFINE which we describe next.

The INIT function instantiates an instance of the VIEWFINDER with a $query$ which a logical representation of a target $W_i \in W$ and a set of materialized views V present in the system. Next, $query$ is assigned to q and each view in V is added to priority queue using OPTCOST(q, v) as the sorting key. At the end of INIT, the candidate views in PQ includes only those views in V.

The PEEK function is used by BFREWRITE to obtain the OPTCOST of the head item in the PQ.

The REFINE function is invoked when BFREWRITE asks the VIEWFINDER to examine the next candidate view. At this stage, the VIEWFINDER pops the head item v out of PQ. The VIEWFINDER then generates a set of new candidate views M by merging v with previously popped candidate views (i.e., views in $Seen$), thereby incrementally exploding the space of candidate views. Note that $Seen$ contains candidate views that have an

OPTCOST less than or equal to that of v. M only retains those candidate that are not already in $Seen$, which are then inserted into PQ. A property of OPTCOST provided as a theorem later is that the candidate views in M have an OPTCOST that is greater than that of v and hence none of these views should have been examined before v. This property enables a gradual explosion of the space of candidate views. Then, v is added to $Seen$.

If v is guessed to be complete (described in Section 6.1.1), we try to find rewrites of q using v by invoking the REWRITEENUM function (described in Section 6.1.2). Among the rewrites found by REWRITEENUM, the cheapest rewrite is returned to BFREWRITE as the result.

6.1.1 Determining Partial or Complete solutions

To determine if a view v is partial or complete with respect to a query q, we take an optimistic approach. This approach represents a *guess* that a complete rewrite exists using v. A guess requires the following necessary conditions as described in [6] (SPJ) and [4] (SPJG) that a view must satisfy to participate in a rewrite of q, although these conditions are not sufficient to confirm the existence of an equivalent rewrite using v.

(i) v contains all the attributes required by q; or contains all the necessary attributes to produce those attributes in q that are not in v

(ii) v contains weaker selection predicates than q

(iii) v is less aggregated than q

The function GUESSCOMPLETE(q, v) performs these checks and returns true if v satisfies the properties i–iii with respect to q. Note these conditions under-specify the requirements for determining that a valid rewrite exists, thus a guess may result in a false positive, but will never result in a false negative.

Algorithm 4 VIEWFINDER

```
function INIT(query, V)
    Priority Queue PQ ←∅; Seen ←∅; Query q
    q ←query
    for each v ∈ V do
        PQ.add(v, OPTCOST(q, v))
    end for
end function

function PEEK
    if PQ is not empty return PQ.peek().OPTCOST else ∞
end function

function REFINE
    if not PQ.empty() then
        v ←PQ.pop()
        M ←MERGE(v, Seen)   ▷ Discard from M those in Seen ∩ M
        for each v′ ∈ M do
            PQ.add(v′, OPTCOST(q, v′))
        end for
        Seen.add(v)
        if GUESSCOMPLETE(q, v) then
            return REWRITEENUM(q, v)
        end if
    end if
return NULL
end function
```

6.1.2 Rewrite Enumeration

In our system the REWRITEENUM algorithm attempts to produce a valid rewrite of a query using a view that is guessed to be complete. The rewrite returned represents the cheapest among all possible equivalent rewrites of q using v. The cost of a rewrite is evaluated by the COST function, and corresponds to the cheapest execution *plan* that implements the rewrite. Equivalence is determined by ensuring that the rewrite and query contain the same attributes, filters, and group-by.

We enumerate equivalent rewrites of q by applying *compensations* [20] to a guessed to be complete view v using L_R. We do this by generating all permutations of required compensations and testing for equivalence, which amounts to a brute force enumeration of all possible rewrites given L_R. This makes case for the system to keep $|L_R|$ small. When L_R is restricted to a known, fixed set of operators it may suffice to examine a polynomial number of rewrites attempts, as in [5] for the specific case of simple aggregations involving group-bys. Such approaches are not applicable to our case as the system should have the flexibility of extending L_R with UDFs from L_W when it results in overall system benefit.

Given the computational cost of finding valid rewrites, BFREWRITE limits the invocation of REWRITEENUM algorithm using two strategies. First, we avoid having to apply REWRITEENUM on every candidate view making a guess for the completeness of a view based on the three properties described earlier. Second, we delay the application of REWRITEENUM to every complete view by determining a lower bound on the cost of a rewrite using v should one exist. For the lower bound we use the OPTCOST, which is described in the next section.

6.2 Determining OPTCOST

The utility of a lower bound we develop in this section enables the enumeration of the candidate views based their potential to provide a low cost rewrite. OPTCOST relies on the non-subsumable cost property of the COST function to arrive at a lower-bound.

Given that v is guessed to be complete with respect to q, a set difference between the attributes, filters and group-bys representation of q and v is referred to as the *fix*. Fix denotes a hypothetical local function that can transform v's representation into q's. Note that a UDF containing such a local function may not really exist. We have to invoke REWRITEENUM which produces a rewrite containing compensations from L_R. The composition of the local functions in the compensation transforms v's representation to q. Finally, note that the existence of fix guarantee that v will result in a valid rewrite for the same reason that guessed to be complete can result in a false positive. Both assume that the required compensation operations can be applied independently of each other to v.

We now describe a OPTCOST function with the two properties that it is a lower bound on the cost of *any* plan returned by REWRITEENUM(q, v) and inexpensive to compute. If v is a materialized view then c_1 is equal to the cost of accessing v. Otherwise, if v results from the merging of views, then c_1 is the total cost to access the constituent views of v. We denote c_2 as the cost of merging the constituent views in v (i.e., creation cost) if v is already not materialized, else $c_2 = 0$ if it is already materialized. We denote c_3 as the cost of applying the least expensive operation in the fix on v, obtained by invoking the COST to obtain the cost of performing each of the operations in the fix on v. c_3 is obtained by $\min(\text{COST}(x, v))$ such that x is an operation in *fix*.

The OPTCOST of v with respect to q is given by: $c = c_1 + c_2 + c_3$, where c is less than the cost of any plan of the rewrite using v. If v is partial with respect to q, then $c_3 = 0$ since no compensation should be applied.

Suppose that the optimizer can generate plans where some of compensations can be pushed into the candidate view v before materializing it. In this case, OPTCOST can provide a weaker lower

bound as it can only consider the cost (c_1) of accessing all the constituent views of v plus the minimum cost c'_3 of applying the least expensive operation in the fix on any constituent views of v or on any intermediate view that can be created in the process of creating v. If v is partial with respect to q, then OPTCOST only includes c_1.

We provide the following proof sketch to capture the correctness of the OPTCOST function.

THEOREM 2. *c is a lower-bound on on the cost of any plan yielded by an equivalent rewrite r of q using v if one exists.*

PROOF. Any plan of the rewrite r will have to access all the constituent views in v and materialize it. So, c_1 and c_2 is common to OPTCOST and the cost of any plan of r.

In order to find the lowest cost r, REWRITEENUM applies all permutations of compensation operations to achieve an equivalent rewrite. Regardless of how many operations are used in the compensation, by Definition 1, the cost of applying the compensations has to be at least as expensive as the cheapest operation c_3 in the fix.

Next we consider the OPTCOST function for the case involving push-down of compensations. For this case, both the ordering of the merges of the constituent views of v as well as the applicable compensations are as yet unknown. Our lower-bound holds as it does not make any assumptions about the ordering of the constituent views in v (i.e., by using c_1) as well as the position of any compensation operator in any plan of r (i.e., c'_3) . □

The following theorem describes the OPTCOST property for newly merged candidates in M. This enables Algorithm 4 to generate candidate views as needed to avoid a pre-explosion of the space of all candidates.

THEOREM 3. *The* OPTCOST *of every candidate view in M that is not in $Seen$ is greater than or equal to the* OPTCOST *of v.*

PROOF. The proof sketch is as follows. The theorem is trivially true for $v \in V$ as all candidate views in M cannot be in $Seen$ and have OPTCOST greater than v. If $v \notin V$, it is sufficient to point out that all constituent views of v are already in $Seen$ since they must have had OPTCOST lesser or equal to v Hence all candidate views in M with OPTCOST smaller than v are already in $Seen$, and those with OPTCOST greater than v will be added to PQ if they are not already in PQ. □

7. EXPERIMENTAL EVALUATION

In this section, we present an experimental study we conducted in order to validate the effectiveness of BFREWRITE in finding low-cost rewrites of complex queries. We first evaluate our methods in two scenarios. The *query evolution* scenario (Section 7.2.1) represents a user iteratively refining queries within a single session. This scenario evaluates the benefit that each new query version receives from the opportunistic views created by previous versions of the query. The *user evolution* scenario (Section 7.2.2) represents a new user entering the system presenting a new query. This scenario evaluates the benefit a new query receives from the opportunistic views created by queries of other users. We compare the performance of our algorithm for the user evolution scenario (Section 7.2.3) with a baseline dynamic programming approach that searches at all targets without using OPTCOST. Next, we evaluate the scalability (Section 7.2.3) of our rewrite algorithm in comparison to the dynamic programming approach. We then compare our method to cache-based methods (Section 7.2.4) that can only reuse identical previous results. We show the performance of our method (Section 7.2.5) under a storage reclamation policy that drops opportunistic views. Lastly, as a sanity check (Section 7.2.6) we compare the quality of rewrites produced by our algorithm with a state-of-the-art DBMS.

7.1 Methodology

Our experimental system consists of 20 machines running Hadoop. We use HiveQL as the declarative query language, and Oozie as a job coordinator. UDFs are implemented in Java, Perl, and Python and executed using the HiveCLI. Some UDFs implemented in our system are log parser/extractor, text sentiment classifier, sentence tokenizer, lat/lon extractor, word count, restaurant menu similarity, and geographical tiling, among others.

Hive currently lacks a *what-if* optimizer functionality, which is needed to obtain cost estimates for hypothetical plans. For these reasons, we created a what-if optimizer that uses the cost model developed for MR frameworks in [14] that considers basic data statistics, number of map and reduce tasks, and number of jobs.

Our experiments use the following three real-world datasets totaling over 1TB: a Twitter log containing 800GB of tweets, a Foursquare log containing 250GB of user check-ins, and a Landmark log containing 7GB of 5 million landmarks including their locations. The identity of a social network user (`user_id`) is common across the Twitter and Foursquare logs, while the identity of a landmark (`location_id`) is common across the Foursquare and Landmarks logs.

The metrics we report during evaluation of query performance are execution time in seconds and the amount of data manipulated (read/write/shuffle) in GB. When comparing BFREWRITE to the Dynamic Programming algorithm, we use the following metrics: running time to find the rewrite, the number of candidate views examined during the search for rewrites, and the number of valid rewrites produced. We limit both algorithms from considering more than 4-way joins simply for practical reasons.

7.1.1 Evolutionary Query Workload

Our test workload simulates 8 analysts who write complex queries containing UDFs, representing realistic marketing scenarios for restaurants. Each analyst in our scenario creates 4 versions of a query, representing the query's evolutions during data exploration and hypothesis testing. We use these 32 queries for all of our evaluations. On average, the original plan of a query creates 17 opportunistic materialized views.

We give a high-level description of these queries in Table 1. We modeled our analyst queries after observing query evolutions in three domains – Yahoo! Pipes queries using public web data, Taverna [18] scientific workflows using biological science data, and TPC-DS [19] interactive queries using warehouse data. Common changes between versions that we identified include a parameter change to allow for more or less data in the results, adding a new subgoal to the query, adding a new UDF or replacing operations with a specialized UDF, and incorporating additional data sources to obtain richer results. Each of our query versions are revised in this manner, and each revision includes at least two types of changes.

In the following set of experiments, $A_i v_j$ corresponds to the Analyst i running version j of her query. We describe one Analyst's query in more depth using Example 4.

EXAMPLE 4. *A_2v_2: Find users with affluent friends that tweet positively about food, and visit a lot of pubs.*

(a): EXTRACT from Foursquare log. EXTRACT restaurant name from text. FILTER by check-ins to wine places. GROUPBY user,

Table 1: Eight analyst marketing scenarios used in evaluation

Analyst 1 wants to find influential users who are interested in food for an advertisement campaign. The evolution of this scenario includes increasingly sophisticated interpretations of what it means to be “influential”.
Analyst2 wants to identity 100 “wine lovers” to send them a coupon for a new wine being introduced. This evolution investigates ways of finding suitable users to whom sending a coupon would have the most impact.
Analyst3 wants to start a gift recommendation service where friends can send a gift certificate to a user u. We want to generate a few restaurant choices based on u’s preferences. The evolution in this scenario will investigate how to generate a diverse set of recommendations that would cater to u and his close set of friends.
Analyst4 wants to identify a good area to locate a sports bar. The area must have a lot of people who like sports and check-in to bars, but the area does not already have too many sports bars in relation to other areas. The evolution focuses on identifying a suitable area where there is good interest but low density of sports bars.
Analyst5 wants to give restaurant owners a customer poaching tool. For each restaurant r, we identify customers who go to a “similar” restaurant in the area but do not visit r. The owner of r may use this to target advertisements. The evolutionary nature focuses on determining “similar” restaurants and their users.
Analyst6 tries to find out if restaurants are losing loyal customers. He wants to identify those customers who used to visit more frequently but are now visiting other restaurants in the area so that he can send them a coupon to win them back. The evolutionary nature of this scenario will focus on how to identify prior active users.
Analyst7 wants to identify the direct competition for each restaurant. He first tries to determine if there is a more successful restaurant of similar type in the same area. The evolutionary nature focuses on identifying what customers like about the menu, food, service, etc. that makes these restaurants successful.
Analyst8 wants to recommend a high-end hotel vacation in an area users will like based on their known preferences for restaurants, theaters, and luxury items. The evolutionary nature focuses on matching user’s preferences with the types of businesses in an area.

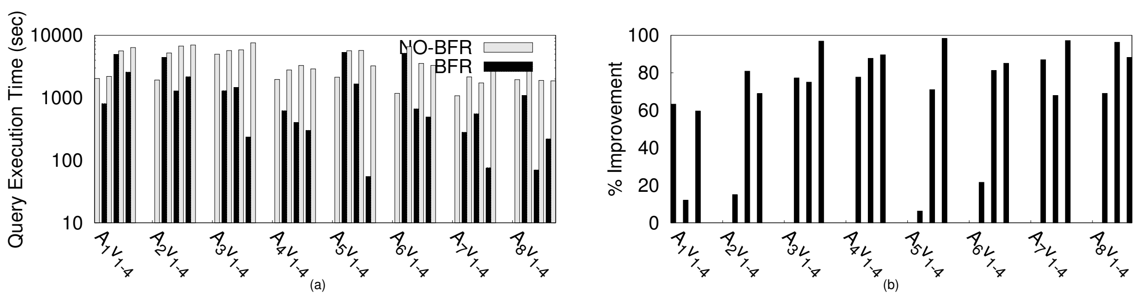


Figure 3: Query evolution comparisons for (a) query execution time (log-scale) and (b) execution time improvement

compute count. Find MAX user check-in. Compute check-in score for each user.

(b): EXTRACT user from Twitter log. Compute FOOD-SENTIMENT score for each user.

(c): CLASSIFY user’s tweets from Twitter log as affluent or not. GROUPBY affluent users, compute affluent tweet count. Find MAX count among all users. COMPUTE an affluent score for each user.

(d): Create social network from Twitter log using tweet source and dest. GROUPBY user pair in social network, count tweets. Find MAX tweets between a user pair. Assign friendship strength score to each user pair.

JOIN (a), (b), and (c) on user_id. Then JOIN with (d) based on friend_id . Threshold based on checkin score, food-sentiment score, friendship score, and affluent score.

7.2 Experimental Results

7.2.1 Query Evolution

This section presents results for the query evolution scenario, where a user’s new query can benefit from opportunistic views created during execution of previous versions of the query. In this experiment, an analyst (e.g., A_i) executes the first version of his query (e.g., $A_i v_1$) followed by issuing each subsequent version (e.g., $A_i v_2$, $A_i v_3$, $A_i v_4$). Subsequent versions of a query are rewritten given the benefit of all opportunistic materialized views from previous versions of the query.

Figure 3(a) shows the performance of the original query plan (NO-BFR in graph) and the optimized query plan after rewriting with BFREWRITE (BFR in graph). Figure 3(b) shows the corresponding percent improvement in execution time for BFREWRITE over the baseline original plan resulting from 10% up to 90% improvement, representing an average improvement of 61% and up to two orders of magnitude. As a concrete data point, $A_5 v_4$ requires 54 minutes to execute in the baseline system, but only 55 seconds after BFREWRITE finds a suitable rewrite. The amount of data manipulated (i.e., read/write/shuffle bytes) closely followed the same trend in Figure 3(a). The significant savings in execution time for BFREWRITE can be attributed to moving much less data, and avoiding the costly re-extraction of data from raw logs whenever possible, due to our algorithm’s ability to reuse prior computation from all previous query versions, i.e., opportunistic materialized views.

7.2.2 User Evolution

This section presents results showing the benefit of our methods for the user evolution scenario, where a user benefits from opportunistic views created during execution of other user’s queries. In this scenario, a new analyst arrives and executes the first version of his query given that all of the other analysts have previously executed their queries in the system. Our BFREWRITE algorithm rewrites $A_i v_1$ using the opportunistic materialized views from all other analyst queries.

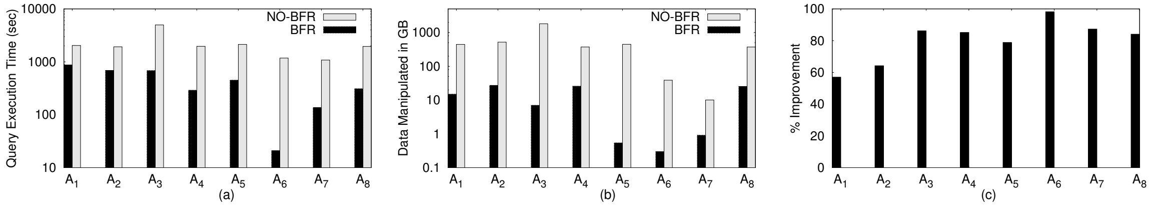


Figure 4: User Evolution Comparisons for (a) query execution time (log-scale)(b) data moved (c) execution time improvement

Figure 4(a) shows the query execution time while Figure 4(b) shows the data manipulated (read/write/shuffle) for BFREWRITE and the original plan. The results demonstrate that query execution time is always lower with BFREWRITE, with a similar trend in the amount of data moved. The percentage improvement in execution time is given in Figure 4(c) where it can be seen that our system results in an improvement of about 50%–90%.

Table 2: Improvement in execution time of A_5v_3 as more analysts become present in the system

Analysts in System	1	2	3	4	5	6	7
Improvement	0%	73%	73%	75%	89%	89%	89%

As an additional experiment, we show the increasing benefit a user may receive each time a new analyst arrives and executes their queries. To show this, first we execute A_5v_3 with no opportunistic views in the system, to create a baseline execution time. We "add" another analyst by executing all four versions of that analyst's query, creating new opportunistic views. Then we re-execute A_5v_3 and report the performance improvement for A_5v_3 over the baseline as new analysts are added. We chose A_5v_3 as it is a complex query using all three logs. Table 2 reports the execution time improvement as analysts are added, showing an increasing benefit as more users are present in the system. The benefit is obviously dependent on similarity to other analysts' queries.

7.2.3 *Runtime Comparison of* BFREWRITE *and DPR*

To evaluate the effectiveness of BFREWRITE to prune the search space of rewrites, we compare it against the baseline DPR algorithm that exhaustively searches the space in order to find the same minimum-cost rewrite. First we evaluate the algorithms during the user evolution experiments, where the only views in the system are those created by the other users. This resulted in approximately 100 views for each user evolution experiment. Next we evaluate the algorithm runtime as we vary the number of views in the system from 1–1000 to show how the algorithms scale with an increasing number of views.

Figure 5 compares the performance of the BFREWRITE and the dynamic programming algorithm (DPR in graph) during the user evolution experiment. While both algorithms find the minimum-cost rewrite r^*, these figures show that BFREWRITE searches much less of the space to find the minimum cost rewrite than does DPR. Candidate views considered in Figure 5(a) corresponds to all the existing views (V) and those views created during the rewrite search process (i.e., merged views). The number of rewrite attempts corresponds to the candidate views examined by REWRITEENUM when searching for a valid rewrite with low cost. These results show for many cases it is possible for BFREWRITE to find r^* by considering far fewer candidate views as shown in Figure 5(a), and orders of magnitude fewer rewrites attempted as shown in Figure 5(b), resulting in significant savings in algorithm runtime as shown in Figure 5(c). This savings is due to BFREWRITE's use of OPTCOST that enables considering the promising candidate views earlier in the search, and exploding the space of candidate views incrementally as needed.

The next experiment evaluates the runtime performance of BFREWRITE and DPR with an increasing number of materialized views in the system. We retained about 9,600 views that were created during the course of design and development of our system, including those created during the process of designing the 8 Analyst scenarios. We use the BFREWRITE and DPR algorithms to find rewrites for one analyst's query (A_3v_1) given a uniform random sampling of subsets of the existing 9,600 views to show how the algorithms scale with the number of materialized views in the system. Furthermore, both the algorithms pruned views at a given target as follows: a view was pruned if it did not reference the same log as the target, if it had a predicate condition preventing it from being used in a rewrite for the target, or if it was a view that is identical to the target. Identical views were removed because those do not help to test the scalability of the algorithms. After pruning, the number of materialized views available for each target was slightly more than 1000. This experiment was performed many times for each algorithm at a given sample size.

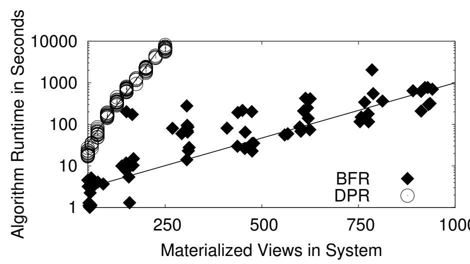


Figure 6: Runtime of BFREWRITE and DPR for varying number of MVs in the system

Figure 6 shows the algorithm runtime for increasing number of materialized views in the system. DPR becomes prohibitively expensive even when 250 MVs are present in the system. BFREWRITE on the other hand scales much better than DPR and has a runtime under 1000 seconds even when the system has 1000 views relevant to the given query. While this runtime is not trivial, we note that these are complex queries involving UDFs that run for thousands of seconds. The amount of time spent to rewrite the query plus the execution time of the optimized query is far less than the execution time of unoptimized queries. For instance, Figure 4(a) reports a query execution time of 451 seconds for A_5 optimized versus 2134 seconds for unoptimized. Even if the rewrite time were 1000 seconds (in this case it is actually 3.1 seconds as seen in Figure 5(c)), the end-to-end execution time would be lower by 11 minutes or 32%.

However, additional pruning techniques [2] can be used to reduce the number of views, thereby reducing the algorithm search space. We note that BFREWRITE will find the minimum cost

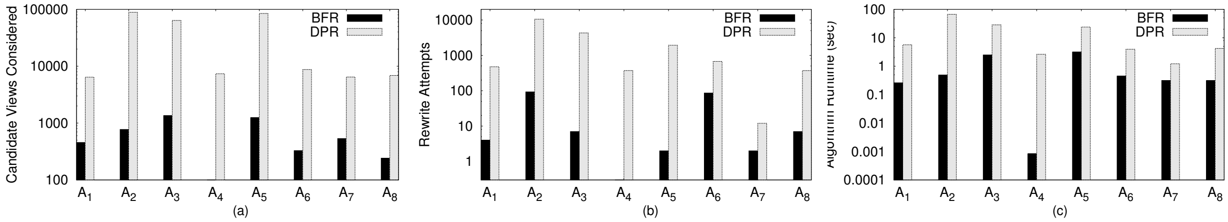


Figure 5: Algorithm Comparisons for (a) Candidate views considered, (b) Refine attempts, and (c) Runtime (log-scale)

rewrite in the space it considers.

7.2.4 *Comparison with Caching-based methods*

Here we compare our approach against caching-based methods such as ReStore [3] that only reuse identical answers (i.e., existing views that require no compensation). Figure 7 shows the query execution time improvements for query evolution of Analyst 1. The results show that both BFREWRITE and ReStore are identical for A_1v_2, while ReStore is worse than BFREWRITE for A_1v_2 and significantly worse for A_1v_3. Reusing cached results can clearly be effective. However, performance is highly dependent upon the presence of identical answers previously generated in the system as cache methods lack a robust capability to reuse previous results.

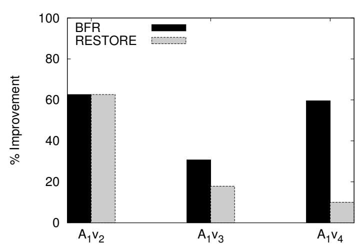


Figure 7: Execution time improvement for query evolution using BFREWRITE **and ReStore**

Table 3: Improvement in execution time of user evolution using BFREWRITE **without identical views**

Analyst	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
BFREWRITE	57%	64%	83%	85%	51%	96%	88%	84%

To illustrate the robustness of our rewrite algorithm beyond caching-based methods, we present an experiment where we prune away all identical views from the solution space. Removing all identical views represents the worst-case scenario for caching-based methods. Such methods will not be able to find *any* rewrites, resulting in 0% improvement. Table 3 reports the percentage improvement using BFREWRITE when identical views have been pruned from consideration. Our algorithm results in very good performance improvement, and these results are comparable to the results in Figure 4(c) that represents the same experiment without pruning identical views. Table 3 shows a performance drop for A_5 compared to the results reported in Figure 4(c) for A_5. This is because A_5 had previously benefited from an identical view corresponding to restaurant similarity which it has to recompute. Note that identical views constituted only 7% of the view storage space in this experiment. Given that analysts pose different but related queries in an evolutionary scenario, any method that relies solely on identical matching can have limited benefit.

Table 4: Execution Time Improvement after reclaiming 10% to 90% of the view storage space

Reclaimed Storage	10%	20%	30%	50%	70%	90%
Improvement	89%	88%	86%	82%	58%	30%

7.2.5 *Storage Reclamation*

Since storage is not infinite a reclamation policy is necessary. We demonstrate the robustness of our approach using unsophisticated policies, since reclamation is not the focus of this work. The first simple policy randomly drops views to reclaim 10%–90% of the view storage space. After each reclamation, we run the A_5v_3 query (used in Section 7.2.2) after applying our rewrite algorithm, and compare it to a baseline without our rewrite algorithm. Each test is repeated twice (different randomizations). Table 4 reports the average execution time improvement for A_5v_3 after reclaiming various amounts of storage space. Table 4 shows that our system is able to find good rewrites for A_5v_3 using the remaining views available, until most of the storage space is reclaimed.

Second, Table 3 provides another perspective on how storage reclamation can affect the effectiveness of BFREWRITE. Specifically, we observe that BFREWRITE is able to find good rewrites even if the optimal views for each query are removed. These results suggest that our approach is robust to "bad" choices made by the reclamation policy, provided some good views remain in the system.

The hardness of designing a good policy is equivalent to the view selection problem with a storage constraint. Certainly better methods [13] could be applied but we leave this for future work.

7.2.6 *Comparison with DB-X*

To verify that our system produces good rewrites, we compared our rewrites to those of a widely-used commercial database system (DB-X) using data from the TPC-H [19] benchmark. DB-X was tuned to its highest optimization level as well as set to consider all views for query rewriting. We created two materialized views on the `LINEITEM` table: group-by–count on `L_ORDERKEY, L_SUPPKEY`, and group-by–count on `L_ORDERKEY, L_PARTKEY`. To test the rewrites produced by both systems, we use three queries on the `LINEITEM` table: a) groupby on `L_ORDERKEY, L_SUPPKEY, L_PARTKEY` b) group-by on `L_ORDERKEY` with count aggregate, and c) group-by on `L_ORDERKEY` with count and max aggregate on `L_SUPPKEY`. The resulting rewrites were given to the DB-X query optimizer, which provided a cost estimate for each rewrite.

BFREWRITE made use of both materialized views for the first query while DB-X did not make use of them which resulted in a rewrite that was 3× worse than the rewrite produced by BFREWRITE. For the second query both BFREWRITE and DB-X resulted in identical rewrites. For the third query, DB-X did not make use of a materialized view, whereas BFREWRITE did re-

sulting in a 5× improvement over DB-X. BFREWRITE produces rewrites competitive to a state-of-the-art commercial DBMS on queries that are much simpler than our setup. Using the DBMS to rewrite the complex queries in our workload would most likely result in very little improvement.

8. RELATED WORK

Query Rewriting Using Views. As noted in Section 6 there has been much previous work on rewriting queries using views. MiniCon [16] and [9] both consider rewriting a single query using existing materialized views, while [8] extends [9] to consider multiple queries. MiniCon extends the bucket-algorithm to be more general, and [9] uses a graph approach and applies a form of subgraph matching among queries and views. In these case both the queries and views are restricted to the class of conjunctive queries (SPJ). Furthermore these methods consider maximally contained rewrites as their context is data integration. Our method considers SPJGA queries and UDFs and our context is query optimization thus we are interested in equivalent rewrites, not maximally contained rewrites. [16] discusses extensions to cost-based rewriting and notes that optimal rewrite for their case is also exponential in the number of views present in the system.

Online Index Selection. Methods such as [10] adapt the physical configuration to benefit a dynamically changing workload by recommending a set of indexes/views to create or drop. Such approaches are not directly applicable to our scenario since materialized views are created simply as artifacts of query execution. View selection methods could be applicable during storage reclamation to retain only those views that provide maximum benefit within a given space budget constraint.

Reusing Computations in MapReduce. Other methods for optimizing MapReduce jobs have been introduced such as those that support incremental computations [12], sharing computation or scans [14], and re-using previous results [3]. As shown in Section 7.2.4, caching-style sharing methods are effective but provide limited benefit compared to our approach.

Multi-query optimization (MQO). The goal of MQO [17] (and similar approaches [14]) is to maximize resource sharing, particularly common intermediate data, by producing a scheduling strategy for a set of queries in-flight. Our work produces a low-cost rewrite rather than a scheduling policy for concurrent query plans.

9. CONCLUSION

We presented a method to takes advantage of opportunistic materialized views to significantly speedup queries in a large-scale data analytics system. With the aid of a UDF model and a lower-bound OPTCOST function we developed in this paper, the BFREWRITE algorithm produces the optimal rewrite while being work efficient. We presented 8 evolutionary queries for realistic scenarios and demonstrated dramatic performance improvements with an average of 61% and up to two orders of magnitude.

Because storage is not infinite, future work will address the problem of identifying the most beneficial views to retain. We will look at view retention strategies from the point of overall system benefit considering these decisions are affected by view maintenance costs, which is an aspect we did not address in this paper.

10. REFERENCES

[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pp. 254–263, 1998.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pp. 496–505, 2000.

[3] I. Elghandour and A. Aboulnaga. ReStore: reusing results of MapReduce jobs. *VLDB*, 5(6):586–597, 2012.

[4] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pp. 331–342, 2001.

[5] S. Grumbach and L. Tininini. On the content of materialized aggregate views. In *PODS*, pp. 47–57, 2000.

[6] A. Halevy. Answering queries using views: A survey. *VLDBJ*, 10(4):270–294, 2001.

[7] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *TODS*, 28(4):517–580, 2003.

[8] G. Konstantinidis and J. Ambite. Optimizing query rewriting for multiple queries. In *IIWeb*, 2012.

[9] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: a graph-based approach. In *SIGMOD*, pp. 97–108, 2011.

[10] K.Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *SMDB*, pp. 459–468, 2007.

[11] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*, pp. 95–104, 1995.

[12] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using MapReduce. In *SIGMOD*, pp. 985–996, 2011.

[13] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Rec.*, 41(1):20–29, 2012.

[14] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *VLDB*, 3(1-2):494–505, 2010.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pp. 1099–1110, 2008.

[16] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB*, 10(2):182–198, 2001.

[17] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[18] Taverna Workflow Management System. http://www.taverna.org.uk.

[19] Transaction Processing Performance Council. TPC-H and TPC-DS Benchmarks.

[20] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD*, pp. 105–116, 2000.